ELSEVIER

Journal of Computational and Applied Mathematics 50 (1994) 221–232

JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS

Parallel algorithms for solving large linear systems

T.J. Dekker, W. Hoffmann *, K. Potma

Department of Computer Systems, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, Netherlands

Received 29 May 1992; revised 21 December 1992

Abstract

The solution of linear systems continues to play an important role in scientific computing. The problems to be solved often are of very large size, so that solving them requires large computer resources. To solve these problems, at least supercomputers with large shared memory or massive parallel computer systems with distributed memory are needed.

This paper gives a survey of research on parallel implementation of various direct methods to solve dense linear systems. In particular are considered: Gaussian elimination, Gauss–Jordan elimination and a variant due to Huard (1979), and an algorithm due to Enright (1978), designed in relation to solving (stiff) ODEs, such that stepsize and other method parameters can easily be varied.

Some theoretical results are mentioned, including a new result on error analysis of Huard's algorithm. Moreover, practical considerations and results of experiments on supercomputers and on a distributed-memory computer system are presented.

Key words: Gaussian elimination; Gauss–Jordan; Linear systems; LU factorization; Pivoting strategies; Parallel algorithms; Vector computing

1. Introduction

The solution of linear systems continues to play an important role in scientific computing. The problems to be solved often are of very large size, so that large computer resources, in particular supercomputers with large shared memory, or massive parallel computer systems with distributed memory are needed to solve them. This appears, for instance, from a survey on very large dense systems solved on high-performance computers given in [10]. In that survey the largest full matrix LU factorization reported by the respondents was of order 55 296, which

* Corresponding author.

SSDI 0377-0427(93)E0131-5

took 4.4 days on a CM-2 machine. Other experiments [13] concern a simulation of elastic light scattering requiring the solution of a large system of complex linear equations. This system has a full complex symmetric matrix which is too large to be kept in the total distributed memory of the computer system. It is solved by means of a parallel implementation of Conjugate Gradients.

This paper deals with parallel implementation of various direct methods to solve dense linear systems, namely Gaussian elimination and some related elimination methods.

Gaussian elimination still seems to be the most popular method to solve dense linear systems. It requires roughly $\frac{2}{3}n^3$ floating-point operations and can be organized such that it performs well on vector processors and parallel systems. Gaussian elimination has been treated in view of implementation on vector and parallel architectures in [14,15,22]. To obtain numerical stability, one mostly uses partial pivoting, which is very satisfactory in practice and can be included without severe performance degradation. Complete pivoting, on the other hand, is seldom used, although it is needed to guarantee numerical stability, as was shown in [26]. It requires more work and a substantial communication overhead on distributed parallel systems. A good compromise can be obtained by using partial pivoting with monitoring, due to [2]. This yields either a guarantee that the pivot growth has been modest, so that accurate LU factorization and solution have been obtained, or a warning that the pivot growth may become too large, in which case transition to complete pivoting may be required. The partial pivoting with monitoring only requires a little extra overhead on a distributed-memory system, as appears from experiments in [17].

Two other elimination algorithms which reduce the given matrix not to triangular but to *diagonal* form (or the identity matrix) are the following. Gauss–Jordan elimination requires about n^3 floating-point operations, which is 50% more than Gaussian elimination. It is, however, very suitable for vector processors, because it operates on full n-vectors (not decreasing in length at each elimination step) and it requires the same total number of vector operations as Gaussian elimination. Thus only for very large systems is Gaussian elimination faster on vector computers [14]. Another variant due to Huard [18], also treated in [3], yields the same result as Gauss–Jordan, but surprisingly requires only about $\frac{2}{3}n^3$ floating-point operations, namely the same as for Gaussian elimination. Implementation of this variant on a vector computer yielded good results; the method is slightly slower than Gaussian elimination, but may require less cache memory on average [16,21].

The numerical solution of stiff systems of ordinary differential equations is mostly done by means of (semi-)implicit methods of multistep or Runge–Kutta type. These implicit methods require the solution of a linear system, where the matrix typically has the form $W = J - (h\alpha)^{-1}I$, J being the Jacobian of the system, h the stepsize and α a scalar depending on the method. Mostly one wants to solve several systems with the same J but different values of (α and) h. In this situation it is attractive to have a decomposition of the matrix which can be easily adapted to these different values. To achieve this, Enright [11] developed an algorithm where the matrix is decomposed using a similarity transformation to Hessenberg form. This algorithm combined with a parallel variant of a diagonally implicit Runge–Kutta method of [25] has been implemented and some preliminary results of experiments are reported.

We first consider parallel computer systems and basic linear algebra subprograms, and subsequently deal with the various methods mentioned to solve dense linear systems.

2. Parallel computer systems and basic linear algebra subprograms

Parallel computer systems can be subdivided into *shared-memory* and *distributed-memory* systems. This is the most important subdivision in view of numerical computation, especially computation requiring large amounts of data as in numerical algebra problems. Extensive surveys are given in [8,12].

Shared-memory systems are supercomputers (and "mini-supercomputers"), which have relatively few (say 4–16) vector processors operating in parallel. Access to the shared memory can only take place simultaneously if different memory banks are addressed. Otherwise a so-called *memory bank conflict* causes a certain delay. Usually successive elements of vectors and columns (in FORTRAN) or rows (in C) of matrices are stored in different banks, so that one vector processor performs stride-one vector operations without memory bank conflicts. If several processors are operating, however, these conflicts are not easily avoidable, and are reduced only to some extent by means of higher level storage such as cache. Thus the memory-access restrictions limit the number of processors, if one wants to avoid a severe overall performance degradation.

Distributed-memory systems have the advantage that each processor can access its own local memory without delay and without conflict with other processors. Thus the number of processors is practically unlimited, and so-called *massive parallel* systems are possible and exist. These systems need a good network topology between the processors and a reasonably fast interprocessor communication speed. Some well-known topologies important for matrix and vector operations are one-dimensional structures (linear array or ring of processors), where the processors are nodes of order 2, two-dimensional structures (mesh, cylinder or torus connection), where the processors are nodes of order 4, and hypercube structures, where the number p of processors is a power of 2 and each processor a node of order $\log_2 p$. In particular, transputers have four communication links for each processor, so that they are suitable to implement two dimensional structures. Matrices can in a natural way be distributed block-wise in a mesh-connected system. For some algorithms a linear array or ring-connected system may, however, be more suitable. The hypercube structures contain the ring connection as well as the torus (or mesh) connection as substructures. These are obtained by ordering the nodes according to a Gray code sequence or according to the combination of two orthogonal Gray code sequences. This was used in [23], where it is concluded that "the hypercube topology does not seem to do any *better* than the grid topology for Gaussian elimination". On the other hand, the communication distance between processors is of order $\log_2 p$ in a hypercube structure, which for large p is considerably smaller than the distance $2(p^{1/2}-1)$ in a mesh-connected system.

In the future the great distinction between shared and distributed memory might disappear or diminish. "It is our belief", says [8], "that parallel processing today is heading in the direction of a shared address space where the physical memory is distributed".

Basic Linear Algebra Subprograms (BLAS) have been designed to enhance portability of numerical software. Three levels of BLAS have been defined in [6,7,19]. Level 1 is a set of vector operations, level 2 a set of matrix–vector operations, and level 3 a set of matrix operations. The latter two levels have especially been designed to enhance performance on

vector and parallel computer systems. In the sequel we shall indicate the main BLAS operations used by the algorithms considered.

3. Gaussian elimination

For given nth-order matrix A and right-hand side vector b, we want vector x, solving the linear system

$$Ax = b. \tag{3.1}$$

We first consider the three parts of Gaussian elimination, namely LU factorization and forward and back substitution, subsequently we deal with pivot selection and strategies, and finally discuss implementation on supercomputers and distributed-memory systems.

LU factorization

The pivoting in its most general form entails both row and column interchanges. Let P denote the permutation matrix of the row interchanges and Q the permutation matrix of the column interchanges. Then the LU factorization with pivoting can be described as: *find lower triangular L, upper triangular U and suitable permutation matrices P and Q such that*

$$PAQ = LU. \tag{3.2}$$

Thus the LU factorization can be considered as starting from matrix $A^{(1)} = PAQ$, i.e., we may describe the process as if the row and/or column permutations have been performed beforehand, and the pivot in each elimination step is in the proper diagonal position. Matrix $A^{(1)}$ is then reduced to an upper triangular matrix $U = A^{(n)}$, by means of $n-1$ successive elimination steps as follows. At the kth elimination step, $k = 1, \ldots, n-1$, matrix $A^{(k+1)}$ is calculated such that the elements in its first k columns below the main diagonal are zero. Let δ_k denote the kth pivot element $A^{(k)}_{k,k}$, let $A^{(k)}_{k,:}$ denote the kth row of $A^{(k)}$, and let $m^{(k)}$ be the column vector whose first k elements are zero and whose remaining elements are given by

$$m^{(k)}_i = A^{(k)}_{i,k}, \quad i = k+1, \ldots, n. \tag{3.3}$$

Then the kth elimination step obtains $A^{(k+1)}$ from $A^{(k)}$ according to

$$A^{(k+1)} = A^{(k)} - \delta_k^{-1} m^{(k)} A^{(k)}_{k,:}. \tag{3.4}$$

This operation is a *rank-one modification*, a level 2 BLAS operation [7]. The final result consists of upper triangular matrix $U = A^{(n)}$ and unit lower triangular matrix

$$L = I + \left(\delta_1^{-1} m^{(1)}, \ldots, \delta_{n-1}^{-1} m^{(n-1)}, 0\right), \tag{3.5}$$

i.e., for $k = 1$ to $n-1$ vector $\delta_k^{-1} m^{(k)}$ is the kth column of matrix $L - I$. Usually the nonzero parts of U and $L - I$ are overwritten on (a copy of) matrix A.

In fact, the diagonals of L and U can here be chosen in different ways. The most general form of the decomposition is LDU, where D is an appropriate diagonal matrix [14]. For simplicity, we have taken $D = I$ (i.e., the identity matrix) and have chosen L (as usual) *unit* lower triangular, i.e., the main diagonal elements of L are equal to one.

Forward substitution

Right-hand side vector b is transformed into a vector y, in correspondence with the LU factorization, i.e., y is calculated as the solution of the triangular system

$$Ly = Pb, \tag{3.6}$$

where Pb is obtained from b by interchanging the elements of b corresponding to the row interchanges of the LU factorization. The forward substitution can be executed in $n-1$ elimination steps, either by means of inner products of rows of L and vector y, or in the form of column operations akin to the elimination steps of the LU factorization as follows. Starting from $b^{(1)} = Pb$, the kth step, $k = 1, \ldots, n-1$, calculates $b^{(k+1)}$ from $b^{(k)}$ according to

$$b^{(k+1)} = b^{(k)} - \delta_k^{-1} b_k^{(k)} m^{(k)}. \tag{3.7}$$

This is a *vector update* operation belonging to the set of BLAS of level 1 [19]. Thus, after $n-1$ steps the solution vector $y = b^{(n)}$ is obtained.

Back substitution

From (3.2) and (3.6) it follows that LU factorization and forward substitution reduce the given linear system (3.1) to the equivalent system

$$UQ^{-1}x = y, \tag{3.8}$$

i.e., first the upper triangular system $Uw = y$ is solved, followed by calculating $x = Qw$. The solution of the triangular system can again be obtained by means of either inner products of rows of U and vector w, or using *vector update* operations as follows. Let $\ddot{U}$ denote the strictly upper triangular part of U and $\ddot{U}_{.k}$ the kth column of $\ddot{U}$. Starting from $y^{(n)} = y$, the kth step, $k = n, n-1, \ldots, 1$, calculates w_k and vector $y^{(k)}$ according to

$$w_k = \delta_k^{-1} y_k^{(k)}, \qquad y^{(k-1)} = y^{(k)} - w_k \ddot{U}_{.k}, \tag{3.9}$$

Finally, the solution vector $x = Qw$ is obtained by interchanging the elements of w, to compensate for the column interchanges in the LU factorization. This means that the elements of the solution vector are interchanged *in reverse correspondence* to the column interchanges in the LU factorization.

Pivot selection strategies

Gaussian elimination is mostly performed with pivoting, i.e., selecting a matrix element of sufficiently large magnitude as pivot of the elimination process, and correspondingly interchanging rows and/or columns of the matrix to bring the pivot in proper position. The pivoting is practically always needed to ensure numerical stability. In direct methods for sparse matrices it is also used to reduce fill-in in the factor matrices L and U, see [9].

In the kth elimination step, an element of largest (or sufficiently large) magnitude is selected in a certain part of $A^{(k)}$. This element will be the kth pivot δ_k. Subsequently the pivot is brought into the proper (k, k)-position by appropriate row and/or column interchanges. For partial pivoting with *row* interchanges (then $Q = I$), the kth pivot is selected in the kth *column* of $A^{(k)}$ on or below the main diagonal. For partial pivoting with *column* interchanges (then $P = I$), the kth pivot is selected similarly in the kth *row* of $A^{(k)}$. For complete pivoting, the kth pivot is selected in the lower right $(n-k+1)$th-order submatrix of $A^{(k)}$.

In practice, one mostly performs partial pivoting. For large systems, however, numerical stability can only be guaranteed with complete pivoting, as shown in [26]. Complete pivoting requires about 50% more operations (namely comparisons) than partial pivoting. It is possible, however, to combine partial and complete pivoting as follows. Partial pivoting is performed with *monitoring the pivot growth*, i.e., in each elimination step an upper bound of the pivot growth is calculated and compared with a certain threshold. When this threshold is exceeded, complete pivoting is used in the remaining elimination steps. A careful choice of the threshold parameter ensures that the algorithm practically is as economic as Gaussian elimination with partial pivoting, and as reliable as Gaussian elimination with complete pivoting [2,14].

Implementation on supercomputers

The most important part of the elimination process, in terms of amount of work to be performed, is the rank-one modification (3.4). It lends itself well to efficient execution on vector and parallel processors. The calculation can be done by rows or by columns, which can, in any order, be processed in parallel. Each row or column operation is a vector update ("AXPY") operation, which is a BLAS routine of level 1 [19]. It can be efficiently executed as a vector operation under certain conditions, depending on the machine architecture and the way the matrix is stored. In particular, column operations can be efficiently executed in FORTRAN and row operations in C, because those vectors are stored contiguously in memory. The rank-one modification can also be done block-wise, which may be attractive for large matrices to avoid page faults and extra communication with secundary storage. For these and other reasons, the rank-one modification is included in the set of BLAS of level 2 [7].

Partial pivoting requires both a column and a row operation, one for selecting a pivot, the other for performing an interchange. For instance, column interchanges mostly are efficient in FORTRAN, and the pivot selection in a row can be performed using a gather operation followed by an operation on the gathered vector.

The vector operations in forward and back substitution can also be performed efficiently on a vector computer. For a system with multiple right-hand sides, the modifications of the right-hand sides in the forward and back substitutions again have the form of rank-one modifications and can be executed in parallel.

Implementation on distributed-memory systems

Experiments on a Meiko Computing Surface with 64 processors, performed in [17], show that if the total capacity of the distributed memory is large enough to store the full matrix and right-hand side of the system, then a parallel version of Gaussian elimination is a suitable solution method. Similar results were obtained in [1]. The matrix is distributed over a square mesh of processors in a *block-scattered* way. Here *scattered* means that single elements (or blocks) are distributed wrapped-around in the row and column directions. This is needed to obtain a good load-balance for the LU decomposition; *block-scattered* means that, instead of single elements, blocks of size 4 or 8, say, are distributed in wrapped-around way. This block-scattering reduces the data traffic in the forward and back substitutions, without significantly affecting the load-balance for the LU decomposition, provided the blocks are small. The method uses a *threshold pivoting* strategy, investigated in [24], which yields a good compromise between numerical stability and the reduction of data traffic between local

memories, needed for the row interchanges. Here *threshold pivoting* means that, instead of an element of largest size, a suitable element which is at least a given fraction θ of the largest element is selected as pivot. Threshold pivoting is also used for sparse matrices with the purpose to reduce fill-in [9]. In our case, we use threshold pivoting to select the pivot locally in the processor containing the (k, k)th element, if this is possible within the threshold constraint; thus, interchanging the kth row and the pivotal row can then be locally performed. From experiments it appears, for instance, that for $\theta \leqslant 0.4$, the amount of inter-processor interchanges is reduced by at least 10%. Experiments show that threshold pivoting is stable in practice [24]. The pivot growth appears to behave as $n^{2/3}$. The experiments reported in [17] used matrices of order up to 4096. The performance efficiency for order larger than 1000 ranged from 75 to 98%. In these experiments also a monitoring of the pivot growth was included. The upper bound for the pivot growth obtained appeared to behave as $n^{3/2}$.

4. Gauss–Jordan elimination

The algorithm of Gauss–Jordan transforms matrix A by means of elementary transformations into a diagonal matrix (or into the identity matrix), and performs similar transformations to the right-hand side vector b in order to find the solution vector x. We first consider the basic algorithm without pivoting and then the algorithm with pivoting and the pivoting strategy.

Basic Gauss–Jordan without pivoting

The transformation of A is achieved in n successive elimination steps. Starting from $A^{(1)} = A$, the kth elimination step, $k = 1, \ldots, n$, transforms $A^{(k)}$ into $A^{(k+1)}$ such that the off-diagonal elements in the kth column, not only below but also above the main diagonal, become zero. Thus, after n steps the diagonal matrix $D = A^{(n+1)}$ is obtained. The kth elimination step can be formulated as follows. The pivot of the kth elimination step is $\delta_k = A^{(k)}_{k,k}$, as in Gaussian elimination. Let g_k be the column vector given by

$$g_k = A^{(k)} e_k - \delta_k e_k, \tag{4.1}$$

i.e., the vector obtained from the kth column of $A^{(k)}$ by replacing its diagonal element by zero. Then the kth elimination step, to introduce the required zeros in the kth column of the matrix, consists of premultiplying $A^{(k)}$ by the matrix

$$T_k = I - \delta_k^{-1} g_k e_k^{\mathrm{T}}. \tag{4.2}$$

In other words, $A^{(k+1)}$ is obtained from $A^{(k)}$ by means of the *rank-one modification*

$$A^{(k+1)} = T_k A^{(k)} = A^{(k)} - \delta_k^{-1} g_k A^{(k)}_{k,\cdot}. \tag{4.3}$$

The corresponding transformation of right-hand side vector b proceeds as follows. Starting from $b^{(1)} = b$, the kth elimination step, $k = 1, \ldots, n$, transforms $b^{(k)}$ into $b^{(k+1)}$ according to

$$b^{(k+1)} = T_k b^{(k)} = b^{(k)} - \delta_k^{-1} b_k^{(k)} g_k, \tag{4.4}$$

which is a *vector update* operation.

Thus, the given linear system is transformed into the equivalent system $Dx = y$, which is easily solved by calculating

$$x = D^{-1}y. \tag{4.5}$$

Gauss–Jordan with pivoting

The selection of pivots and the corresponding interchanges take place in a similar way as in Gaussian elimination. In the kth elimination step, $k = 1, \ldots, n$, the kth pivot is selected in the lower right $(n-k+1)$th-order submatrix of $A^{(k)}$, where in principle the same pivoting strategies can be chosen as for Gaussian elimination.

Mostly, partial pivoting is performed, which, as explained in Section 3, can use either row interchanges or column interchanges. The numerical behaviour of the Gauss–Jordan algorithm is quite different, however, for these two strategies, in contrast to the behaviour of Gaussian elimination. The accuracy of the calculated solution is of the same order of magnitude in all cases. The difference manifests itself in the size of the residual $r = b - Ax$ of a calculated approximate solution x. Gauss–Jordan using partial pivoting with *row* interchanges often yields a much larger residual corresponding to the calculated solution than Gaussian elimination does, as shown in [20]. On the other hand, Gauss–Jordan using partial pivoting with *column* interchanges yields a residual which is mostly not larger than the residual obtained by Gaussian elimination of the same system. This follows from an error analysis of [5] which we here briefly explain. The main difference between Gaussian elimination and Gauss–Jordan is that the latter algorithm calculates the LU factorization of the matrix and the explicit inverse of matrix U during the reduction of the matrix to diagonal form. The result of the whole algorithm can be described as follows. Let γ be the growth factor, ϵ the machine precision, Q the permutation matrix for the column interchanges, V the calculated inverse of U, and $\phi_i(n)$ low-degree polynomials in n for $i = 1, \ldots, 4$. We then have the following theorem.

Theorem 4.1. *Gauss–Jordan using partial pivoting with column interchanges exactly satisfies the following relations, where E_i, for $i = 1, \ldots, 4$, are appropriate matrices, E_2 being lower, E_3 and E_4 upper triangular:*

$$\begin{aligned} &LU = AQ + E_1, && \|E_1\| < \phi_1(n)\|A\|\gamma\epsilon, \\ &(L + E_2)y = b, && \|E_2\| < \phi_2(n)\|L\|\epsilon, \\ &VU + E_3 = I, && \|E_3\| < \phi_3(n)\|V\|\epsilon, \\ &(V + E_4)y = w, && \|E_4\| < \phi_4(n)\|V\|\epsilon. \end{aligned} \tag{4.6a}$$

Hence, the calculated solution vector $x = Qw$ satisfies the exact relation

$$(AQ + E_1 + E_2U)(I - E_3 + E_4U)^{-1}w = b. \tag{4.6b}$$

A proof of this theorem is given in [5]. The first two relations of (4.6a) are the same as for Gaussian elimination; the remaining two are different and reflect the calculation of U^{-1} in the Gauss–Jordan algorithm described. Gauss–Jordan can also be performed with complete pivoting or with monitoring the pivot growth, in order to obtain a more reliable algorithm for very large systems.

Vectorization and parallelization aspects

The most important part of the Gauss–Jordan algorithm is the rank-one modification (4.3), which can be performed efficiently on vector and parallel processors in a similar way as for Gaussian elimination. Moreover, the vector operations are more efficient (on the average), because the elimination steps operate on entire columns (one element in each column excepted), whereas the elimination steps in Gaussian elimination operate on columns of lengths decreasing from n to 1. Although Gauss–Jordan requires about 50% more floating-point operations than Gaussian elimination, namely about n^3 operations versus $\frac{2}{3}n^3$ operations, these algorithms require the same number of vector operations, namely $\frac{1}{2}n^2$ vector update operations. Moreover, the rank-one modification can be performed block-wise to avoid page faults for large n, in the same way as for Gaussian elimination.

The results of experiments on numerical stability and timing of this algorithm and a similar algorithm for matrix inversion have been published elsewhere [4,5]. These results show that Gauss–Jordan can be rather efficiently implemented on a supercomputer. Although it requires more work, it is competitive with Gaussian elimination for order n up to nearly 50.

Gauss–Jordan elimination is particularly suitable for calculating the inverse of a matrix. Matrix inversion, by means of Gauss–Jordan or Gaussian elimination, requires about $2n^3$ floating-point operations. Gauss–Jordan matrix inversion can, however, be arranged such that only n^2 vector operations are needed. This cannot be achieved using Gaussian elimination.

5. Gauss–Huard algorithm "méthode des paramètres"

A variant of Gauss–Jordan algorithm introduced by Huard [18], also treated in [3], reduces a given matrix to the identity matrix, i.e., yields the same result as Gauss–Jordan, possibly apart from a different diagonal scaling.

Huard's algorithm requires only about $\frac{2}{3}n^3$ floating-point operations, the same as needed for Gaussian elimination. The algorithm proceeds in n steps, numbered $k = 1, \ldots, n$. Only partial pivoting with column interchanges is possible. Thus, as earlier, we define $A^{(1)} = AQ$ and $b^{(1)} = b$. As an illustration we display matrix and right-hand side vector at the start of the fourth step:

$$A^{(4)} = \begin{pmatrix} 1 & 0 & 0 & x & \cdots & x \\ 0 & 1 & 0 & x & \cdots & x \\ 0 & 0 & 1 & x & \cdots & x \\ a & a & a & a & \cdots & a \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ a & a & a & a & \cdots & a \end{pmatrix}, \qquad b^{(4)} = \begin{pmatrix} x \\ x \\ x \\ b \\ \vdots \\ b \end{pmatrix}.$$

Here a and b denote original elements, and x denotes elements which have been modified in previous steps.

At the start of the kth step, the first $k-1$ rows have been transformed such that the upper left submatrix of order $k-1$ is transformed into the identity matrix. The remaining $n-k+1$ rows are, however, unchanged at this moment, contrary to what happens in Gaussian elimination and Gauss–Jordan. The kth elimination step consists of the following three parts.

(A) *Row elimination*: the first $k-1$ elements of the kth row are eliminated using the first $k-1$ rows; this requires $k-1$ vector updates of vectors of length $n-k+1$, or equivalently, row vector times matrix subtraction from the kth row, i.e., $2(k-1)(n-k+1)$ floating-point operations.

(B) *Row scaling*: subsequently, the kth row is scaled such that its diagonal element becomes one; this takes one division and $n-k$ multiplications.

(C) *Column elimination*: finally, the elements in the kth column above the main diagonal are eliminated, where the kth diagonal element is used as pivot; this final part is quite similar as in Gauss–Jordan elimination; it can be viewed as a rank-one update of the upper-right $(k-1)\times(n-k)$ submatrix, which requires $2(k-1)(n-k)$ floating-point operations.

Thus the amount of floating-point operations in the kth step equals

$$2(k-1)(n-k+1)+(n-k+1)+2(k-1)(n-k)=4(k-1)(n-k)+n+k-1,$$

so that the total amount of operations roughly equals $\frac{2}{3}n^3$.

Pivot selection and strategies

In the Gauss–Huard algorithm only partial pivoting with *column* interchanges is possible, i.e., in the kth step after performing part (A) of the eliminations, the pivot is selected in the kth *row* and, if needed, a corresponding interchange of columns is performed. Gauss–Huard with this pivoting strategy has about the same numerical behaviour and stability as Gauss–Jordan with partial row pivoting explained above. This follows from the following *new* result.

Theorem 5.1. *Gauss–Huard with row pivoting and column interchanges exactly satisfies the following relations, where E_i, for $i=1,\ldots,4$, are appropriate matrices, E_3 and E_4 again being upper triangular, but here both E_1 and E_2 are full matrices*:

$$\begin{aligned} &LU=AQ+E_1, && \|E_1\|<\phi_1(n)\|A\|\,\|V\|\epsilon,\\ &(L+E_2)y=b, && \|E_2\|<\phi_2(n)\|A\|\,\|V\|\epsilon,\\ &VU+E_3=I, && \|E_3\|<\phi_3(n)\|V\|\epsilon,\\ &(V+E_4)y=w, && \|E_4\|<\phi_4(n)\|V\|\epsilon. \end{aligned} \tag{5.1}$$

Hence, the calculated solution vector $x=Qw$ satisfies an exact relation of the form (4.6b).

A proof of this theorem will be given in a future paper.

Note that the calculated results of Gauss–Jordan and Gauss–Huard, although mathematically the same, mostly are numerically different. The main difference is that in Gauss–Huard matrix L is not explicitly calculated, but only implicitly in product form. This explains the saving in number of operations.

The good numerical behaviour and stability of Gauss–Huard was confirmed by experiments in [16,21]. These experiments, both on a Cyber 205 and an Alliant FX/4, showed also that the Gauss–Huard algorithm is competitive in speed with a similar implementation of Gaussian elimination, although slower than the corresponding LINPACK routine. In particular it appears that Gauss–Huard can be arranged such that, for large n, it requires about half the number of page faults (on a Cyber 205) or cache refreshes (on an Alliant) as needed for

Gaussian elimination. This is achieved by storing A by rows (i.e., by storing A^T in FORTRAN). For details, see [21].

6. Updating LU factorization using transformation to Hessenberg form

In the numerical solution of (stiff) ordinary differential equations, it is often required to solve a sequence of linear systems of the form

$$(A + \mu I)x = b, \tag{6.1}$$

for various values of μ. This is needed in (semi-)implicit methods, as well multistep as Runge–Kutta-type methods, see [11,25]. Calculating a new LU factorization for each new value of μ may be rather time-consuming. One can define another factorization of the given matrix which is faster to update for new values of μ. The following algorithm is due to Enright [11].

Decompose the given matrix for $\mu = \mu_1$ by means of the similarity transformation

$$PAP^{-1} + \mu_1 I = LHL^{-1}, \tag{6.2}$$

where H is upper Hessenberg, L is unit lower triangular and P is a permutation matrix for stabilizing row and column interchanges. This decomposition requires about $\frac{5}{3}n^3$ floating-point operations. For other values $\mu = \mu_i$, $i = 1, \ldots, r$, say, we then have

$$PAP^{-1} + \mu_i I = L(H + (\mu_i - \mu_1)I)L^{-1}. \tag{6.3}$$

For each value of μ_i, the Hessenberg matrix is then LU-factorized with row interchanges as follows:

$$P_i(H + (\mu_i - \mu_1)I) = L_i U_i, \tag{6.4}$$

where P_i is a permutation matrix for the row interchanges. As H is upper Hessenberg, matrix L_i is a lower bidiagonal matrix and the LU factorization requires only about n^2 floating-point operations. So, if r is the number of different values of μ, and k the number of different right-hand sides to be treated for each new value of μ, then the total number of floating-point operations to solve (6.1) for all these cases is about $\frac{5}{3}n^3 + r(1 + 3k)n^2$. Hence, for large values of n and modest values of k, this method is faster than ordinary Gaussian elimination if $r > \frac{5}{2}$.

Applying this to diagonally implicit methods for solving (large) systems of differential equations, we take A to be an estimate of the Jacobian of the system, and μ_i equal to $(h\alpha)^{-1}$, where h is the stepsize and α a parameter depending on the method. If the system of differential equations is linear or only mildly nonlinear, then J is or can remain constant during several steps, so that r can be rather large. In that case, Enright's method is advantageous.

Enright's method has been incorporated in a parallel variant of a diagonally implicit Runge–Kutta method due to [25]. Experiments on a CRAY Y-MP showed that a speedup as expected can be achieved. For example, a problem of Davison, mentioned in [11], which is a system of 80 linear differential equations, yielded a speedup by a factor 1.5.

Detailed results will be published in a future paper.

References

[1] G. Bader and E. Gehrke, On the performance of transputer networks for solving linear systems of equations, *Parallel Comput.* **17** (12) (1991) 1397–1407.

[2] P.A. Businger, Monitoring the numerical stability of Gaussian elimination, *Numer. Math.* **16** (1971) 360–361.

[3] M. Cosnard, Y. Robert and D. Trystram, Résolution parallèle de systèmes linéaires denses par diagonalisation, *EDF Bull. Direction Études Rech. Sér. C Math. Inform.* (2) (1986) 67–88.

[4] T.J. Dekker and W. Hoffmann, Numerical improvement of the Gauss–Jordan algorithm, in: A.H.P. van der Burgh and R.M.M. Mattheij, Eds., *Proc. ICIAM 87, Contributions from the Netherlands* (La Villette, Paris, 1987) 143–150.

[5] T.J. Dekker and W. Hoffmann, Rehabilitation of the Gauss–Jordan algorithm, *Numer. Math.* **54** (1989) 591–599.

[6] J.J. Dongarra, J. Du Croz, S. Hammarling and I.S. Duff, A set of level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Software* **16** (1990) 1–17; 18–28.

[7] J.J. Dongarra, J. Du Croz, S. Hammarling and R.J. Hanson, An extended set of Fortran Basic Linear Algebra Subprograms, *ACM Trans. Math. Software* **14** (1988) 1–17; 18–32.

[8] J.J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers* (SIAM, Philadelphia, PA, 1991).

[9] I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices* (Clarendon Press, Oxford, 1986).

[10] A. Edelman, The first annual large dense linear system survey, *SIGNUM Newsl.* **26** (4) (1991) 6–12.

[11] W.H. Enright, Improving the efficiency of matrix operations in the numerical solution of stiff ordinary differential equations, *ACM Trans. Math. Software* **4** (1978) 127–136.

[12] R.W. Hockney and C.R. Jesshope, *Parallel Computers 2* (Adam Hilger, Bristol, 1988).

[13] A.G. Hoekstra, P.M.A. Sloot, M.J. de Haan and L.O. Hertzberger, Time complexity analysis for distributed memory computers: Implementation of a parallel Conjugate Gradients method, in: J. van Leeuwen, Ed., *Proc. Computing Science in the Netherlands, CSN 91* (SION, Utrecht, 1991) 249–266.

[14] W. Hoffmann, Solving linear systems on a vector computer, *J. Comput. Appl. Math.* **18** (3) (1987) 353–367.

[15] W. Hoffmann, Basic transformations in linear algebra for vector computing, Thesis, Univ. Amsterdam, 1989.

[16] W. Hoffmann, A fast variant of the Gauss–Jordan algorithm with partial pivoting, in: W. Hoffmann, Basic transformations in linear algebra for vector computing, Thesis, Univ. Amsterdam, 1989, Chapter IV, 53–60.

[17] W. Hoffmann and K. Potma, Threshold pivoting in Gaussian elimination to reduce interprocessor communication, Technical Report CS-91-05, Dept. Comput. Systems, Univ. Amsterdam, 1991.

[18] P. Huard, La méthode simplex sans inverse explicite, *EDF Bull. Direction Études Rech. Sér. C Math. Inform. 2* (1979) 79–98.

[19] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic Linear Algebra Subprograms for Fortran usage, *ACM Trans. Math. Software* **5** (1979) 308–323; 324–325.

[20] G. Peters and J.H. Wilkinson, On the stability of Gauss–Jordan elimination with pivoting, *Comm. ACM* **18** (1975) 20–24.

[21] K. Potma, Huard's method for solving linear systems on vector and parallel vector computers, Technical Report CS-89-12, Dept. Comput. Systems, Univ. Amsterdam, 1989.

[22] Y. Robert, *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm* (Manchester Univ. Press, Manchester, 1990).

[23] Y. Saad, Gaussian elimination on hypercubes, Res. Report YALEU/DCS/RR-462, 1986.

[24] L.N. Trefethen and R.S. Schreiber, Average-case stability of Gaussian elimination, *SIAM J. Matrix Anal. Appl.* **11** (1990) 335–360.

[25] P.J. van der Houwen and B.P. Sommeijer, Iterated Runge–Kutta methods on parallel computers, *SIAM J. Sci. Statist. Comput.* **12** (1991) 1000–1028.

[26] J.H. Wilkinson, Error analysis of direct methods of matrix inversion, *J. Assoc. Comput. Mach.* **8** (1961) 281–330.